# On-the-fly Translation and Execution of OCL-like Queries on Simulink Models

Beatriz A. Sanchez*, Athanasios Zolotas†, Horacio Hoyos Rodriguez‡, Dimitris S. Kolovos§ and Richard F. Paige**
*Department of Computer Science*
*University of York, York, United Kingdom*
{basp500*, thanos.zolotas†, dimitris.kolovos§, richard.paige**}@york.ac.uk,
horacio.hoyos.rodriguez@ieee.org ‡, paigeri@mcmaster.ca**

***Abstract*—MATLAB/Simulink is a tool for dynamic system modelling. Model management languages such as OCL, ATL and the languages of the Epsilon platform tend to focus on the Eclipse Modelling Framework (EMF), a de facto standard for domain specific modelling. As Simulink models are built on an entirely different technical stack, the current solution to manipulate them using such languages requires their transformation into an EMF-compatible representation. This approach is expensive as the cost of the transformation can be crippling for large models, it requires the synchronisation of the native Simulink model and its EMF counterpart, and the EMF-representation may be an incomplete copy of the model. In this paper we propose an alternative approach that uses the MATLAB API to bridge Simulink models with existing model management languages that relies on the "on-the-fly" translation of model management language constructs into MATLAB commands. Our approach eliminates the cost of the transformation and of the co-evolution of the EMF-compatible representation while enabling full access to the Simulink model details. We evaluate the performance of both approaches using a set of model validation constraints executed on a sample of the largest Simulink models available on GitHub. Our evaluation suggests that the translation approach can reduce the model validation time up to 80%.**

***Index Terms*—Eclipse Modelling Framework, MATLAB Simulink, Model Driven Engineering, Epsilon**

## I. Introduction

MATLAB/Simulink is a modelling tool for dynamic systems that is widely used across many industries such as aerospace and automotive [1, 2, 3]. In model-driven engineering processes, models are queried, transformed, modified, and validated (amongst other activities). Many state-of-the-art modelling management frameworks that support these activities are tailored for models conforming to the Eclipse Modelling Framework (EMF) [4], a de facto standard for domain-specific modelling [5]. Modelling environments that build atop EMF, such as Papyrus [6] and Capella [7], have at their disposal the model management facilities from these frameworks, but this is not the case for MATLAB/Simulink models which are built on an entirely different technical stack.

Some attempts to manipulate Simulink models (e.g. [8, 9, 10]) have resulted in single-use solutions tailored for specific model management activities. A more reusable approach is provided by the Massif project [11] which offers a set of facilities that can transform Simulink models into an EMF-compatible representation and vice-versa. While this solution is more reusable, the cost of the Simulink-to-EMF transformation can be crippling when large Simulink models are involved, as demonstrated later. Evidently, keeping the EMF representation of continuously changing Simulink models synchronised requires the repetitive execution of the transformation procedures whenever the Simulink model or its EMF-counterpart change. Moreover, the current EMF representations of Simulink models are an incomplete copy of the model as, for example, their meta-model does not consider Stateflow blocks.

Given the industry adoption of MATLAB/Simulink, in this paper we propose a new approach to bridge Simulink models with existing model management languages, addressing these issues by generating and executing MATLAB commands on-the-fly from OCL-like queries. Our approach does not require an upfront transformation which eliminates the round trip engineering costs of the transformation and of the co-evolution of the EMF-counterpart. Moreover, through the use of MATLAB's API to resolve model element types, their properties and operations, our solution enables the manipulation of the Simulink complete model.

In this work, we compare the performance of our approach against Massif's upfront-transformation by measuring the execution time of different stages of a validation process. This process involves the execution of OCL-like invariants that validate structural properties on a sample of the largest available Simulink models on GitHub. Our evaluation indicates that our approach is more appropriate for continuously changing models as it can significantly reduce the transformation overhead by reducing the overall time of the validation process by up-to 80%. In contrast, the transformation approach is more convenient for signed-off models that need to be extensively queried as the cost of a transformation is a one-off and the validation overhead 2 orders of magnitude faster.

Our approach is implemented using the Epsilon [12] model management framework; however, the approach and evaluation results are relevant for other frameworks with similar model connectivity facilities, such as ATL.

**Roadmap.** The rest of the paper is structured as follows. Section II introduces the modelling technologies used in our approach and evaluation. Section III presents the architecture of our "live" approach to bridge MATLAB Simulink

models into Epsilon. Section IV evaluates the execution-time performance of both bridge approaches on a sample of large Simulink models. Section V discusses observations and lessons learnt. Section VI summarizes related work. Section VII concludes the paper and discusses future work.

## II. Background

We herby introduce the modelling technologies at the core of this work: MATLAB/Simulink, Epsilon, EMF and Massif.

### A. MATLAB/Simulink

MATLAB is a proprietary programming tool developed by MathWorks that provides a variety of numerical computing environments. Under its Simulink [13] package, MATLAB provides a graphical block-based modelling framework that enables the modelling, simulation and analysis of dynamic systems and supports model management operations like code generation and continuous model verification. Additionally, its Stateflow [14] package adds decision logic to Simulink models through state machines and flow charts that describe how blocks react to input signals, events and time-based conditions.

**Simulink Models.** These are dynamic systems models based on blocks that represent sub-systems and connections between them. Figure 1 presents a sample Simulink model (from [15]) that represents the behaviour of a car in motion after the accelerator pedal is pressed . The model contains five blocks from the Simulink library: a pulse generator, a gain, a second-order integrator and two outports. The pulse generator produces an input signal which simulates the accelerator pedal. The gain simulates the multiplied effect in the car acceleration. The second-order integrator enables the acquisition of the position and speed of the car from the acceleration through its outports. These Simulink blocks are inter connected at their ports through directed lines called signals.

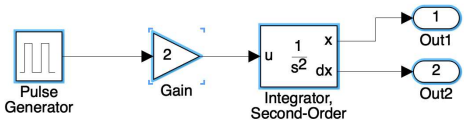


Fig. 1. Example MATLAB/Simulink model.

Simulink models are composed of elements of different type e.g. `Block`, `Line`, `Port`, but also have a specific subtype. For example, an element of *type* `Port` may have an `inport` or `outport` *subtype*. In Figure 1 all highlighted elements are of type `Block` and their subtypes, from left to right, are: `DiscretePulseGenerator`, `Gain`, `SecondOrderIntegrator` and `Outport`.

**MATLAB/Simulink commands.** In addition to MATLAB's graphical interface, Simulink models can be managed through MATLAB/Simulink commands. Listing 1 shows sample functions that enable model navigation and modification. MATLAB/Simulink Models are file based and have to be loaded before interacting with them. Line 1 shows how to load a model named *carModel*. The `find_system` function in line 2 shows how to retrieve all model elements of a given type, in this case: `Block`. Assuming *carModel* refers to the model in Figure 1, this function would return five blocks, and by changing the `Block` parameter for `Line` or `Port` it would return the 4 signals or the 8 ports in the figure. Line 3 illustrates a model query at subtype level which uses the `BlockType` parameter to retrieve specific block elements. Alternatively the parameters `LineType` or `PortType` can be used to collect line or port subtype elements. Applied on the model of Figure 1, the statement in line 3 would return the second block element.

In the same listing, line 4 shows how to create a new model element, in this case, a `Gain` block. The first argument of the `add_block` function is the *path* of the library block to be used, in this case, a Gain block in the Simulink library, while the second argument is the *path* where the new model element should be created. This path starts with the name of the model, ends with the new element's name, and may contain in between the name of intermediary `SubSystem` blocks that will ultimately contain the new element. Lines 5 and 6 of the listing show the use of getter and setter functions. Line 5 shows how to retrieve the block's subtype property and line 6 how to set the block's name.

```
1 load_system carModel
2 blocks=find_system('carModel','FindAll','on','
      Type','Block')
3 gainBlocks=find_system('carModel','FindAll','on',
      'BlockType','Gain')
4 gain=add_block('simulink/Math Operations/Gain','
      carModel/SubSystem/Gain')
5 chartBlockType=get_param(gain,'BlockType')
6 set_param(gain,'Name','newName')
```

Listing 1. MATLAB/Simulink commands.

**MATLAB Java API.** MATLAB provides Application Programming Interfaces (APIs) for languages like C++, Python, C, Fortran and Java. The Java API [16] provides an interface to MATLAB-specific types e.g. structural arrays, and to the MATLAB engine where MATLAB functions can be evaluated. Listing 2 shows sample methods provided by this API. Lines 1 and 5 show how to start and close the connection with the MATLAB engine. Lines 2 and 3 evaluate MATLAB commands on the engine that are passed as strings to the `eval` method. Line 4 shows how to retrieve the value of a variable from the engine, in this case the one declared in line 3.

```
1 MatlabEngine eng = MatlabEngine.startMatlab();
2 eng.eval("load_system model;");
3 eng.eval("m = getSimulinkBlockHandle('model')");
4 Object m = eng.getVariable("m");
5 eng.close();
```

Listing 2. MATLAB Java API

### B. Epsilon

Epsilon is a model management framework that provides a family of inter-operable languages and tools designed for model management tasks like model navigation, validation and transformation. The Epsilon Object Language (EOL) [17] is an OCL-like model query and transformation language that all other Epsilon languages are built on top of. Among these model management languages we find the Epsilon Validation Language (EVL) [18] —designed to evaluate invariants on

model elements, and the Epsilon Transformation Language (ETL) [19] —targeted at model-to-model transformations.

Epsilon has a layered architecture. The Epsilon Model Connectivity (EMC) middle-layer provides abstraction facilities that allow models of arbitrary technologies (e.g. EMF, XML) to be managed in a uniform manner in any of the Epsilon languages. Concrete EMC implementations for different modelling technologies such as EMF, or PTC-Integrity Modeler, are known as EMC drivers. Listing 3 shows an EOL program that can be executed on models of arbitrary modelling technologies due the EMC facilities. Basically, the model (represented by `M` in the script[1]) would be injected to the EOL interpreter at runtime by a specific EMC driver.

```
1 var element = M!Block.all.first();
2 var name = element.name;
3 element.evaluate();
4 var newElement = new M!Block;
5 newElement.name = "My Block";
```

Listing 3. Example EOL Script.

Provided the injected model contains elements of type Block, line 1 of the previous listing shows how to retrieve all elements of this type from the model using the `all` keyword and later on how to select the first element of the collection using the `first()` operation. The `all` keyword calls a method, implemented by the EMC driver, which collects all elements of the preceding type, in this case `Block`. In contrast, the operation `first()` is provided by default by EOL and works on collections. Other operations such as `select()` and `collect()` are provided in EOL by default, along with other language constructs like if statements and for loops. Line 1 additionally shows how to declare and assign the value returned by `first()` to the `element` variable. Line 2 shows how to retrieve the value of the `name` property from the `element` variable while line 3 shows how to invoke the `evaluate()` method on the same block element. Further down, line 4 shows how a new element of type `Block` is created and assigned to the `newElement` variable while line 5 sets the `name` property on this element.

The syntax that an EOL program uses to create and delete model elements, to set and get their properties, or invoke their methods does not depend on the EMC driver. The contribution of an EMC driver to the script is the availability of model element types, their properties and additional methods. For Listing 3 to terminate successfully, the EMC driver used at runtime to provide model `M` would need to manage model elements of type `Block` with a `name` property and an `evaluate()` method. Some of the modelling abstractions that EMC drivers implement to achieve this are presented in the top compartment of Figure 3 and will be discussed in section III.

Epsilon currently provides EMC drivers for a variety of modelling technologies including EMF, XML [17] and Spreadsheets [20]. Section III presents the architecture of the new Simulink EMC driver which is the main contribution of this work.

[1]The character "!" is used in Epsilon to separate the runtime name of the model from the *type* or *kind* of the model element.

### C. Eclipse Modelling Framework and Massif

The Eclipse Modelling Framework (EMF) was originally designed to build Java applications based on domain-specific model definitions. The meta-modelling language used to describe EMF models is Ecore. EMF offers several representations of Ecore models including Java code, XML Schema, and UML diagrams, but its canonical format is the XML Metadata Interchange (XMI).

**Massif.** The Massif [11] project enables the transformation of MATLAB/Simulink models into an EMF-compatible representation and vice-versa. The resulting EMF models conform to an Ecore Simulink meta-model defined by the project. Massif connects to MATLAB's engine in order to parse or write Simulink models. The project's facilities that transform a MATLAB/Simulink model into EMF or vice-versa result in partial model-to-model transformations as they are limited to Simulink elements and not Stateflow elements.

**Massif's Simulink Ecore meta-model.** The Massif meta-model considers any Simulink model element that can be identified and named as a subtype of the `SimulinkElement` class and their identity is stored as an element of class `SimulinkReference` [21]. All subclasses of `SimulinkElement` are presented in Figure 2. Direct subtypes of this class are `Block`, `Port`, `Connection` and `SimulinkModel`. The `SimulinkModel` class is the root model element which keeps a reference to the file and version of the original MATLAB/Simulink model. This class contains all the `Block` elements along with their `Port` and `Connection` elements.

In Massif, the ports (`Port`) of a block are either of type `InPort` or `OutPort` and they can be represented by a virtual block of class `PortBlock`. Similarly, the lines that connect the block ports are instances of the `Connection` class which can be either `SingleConnection` or `MultiConnection`. Any block whose MATLAB subtype can't be found as a class in Massif is considered as a generic `Block`. Some blocks have predefined properties as attributes e.g. the `tag` property in the `SubSystem` class, but most of their properties are dynamically added to their `parameters` attribute which contains array of `Property` elements, each with a specific name, value and type.

Some of the Massif meta-model constructs differ from the way MATLAB manages Simulink models. The most notable difference is that MATLAB/Simulink's block library offers 140 different `Block` subtypes (e.g. Gain, Sum, UnitDelay, etc.) while Massif only provides 11 concrete ones. The MATLAB/Simulink subtype of blocks that do not fall under the previous 11 subtypes can be retrieved from the block's `parameters` attribute, looking for the one with the `BlockType` identifier. Similarly, there are 5 `Port` subclasses in Massif's meta-model out of the 6 subtypes found in the MATLAB/Simulink library and, in particular, it is unclear how the `State` class in Massif maps to one or both of the `Reset` and `ifaction` port types in Simulink. A related inconsistency happens when, after a transformation into EMF, the attributes of some block subclasses can have redundant or unpopulated values as they can also be found within the block's `parameters`

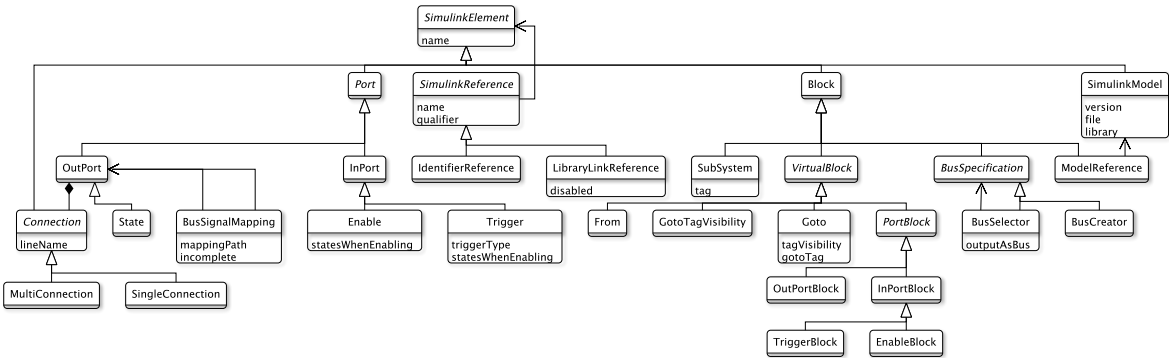


Fig. 2. Simulink element types provided by Massif's Simulink meta-model.

attribute e.g. the `tag` attribute in the `SubSystem` class which can also be found in the `parameters`. Another difference is that the Massif `Connection` class refers to MATLAB/Simulink elements of type `Line` and subtype `signal` and that the `MultiConnection` and `SingleConnection` subclasses in the meta-model are used refer to the `SegmentType` property of lines in MATLAB which can take the value of `trunc` or `branch`, correspondingly. In addition, in MATLAB/Simulink commands subtype capitalization is important e.g. `input` is used to refer to a the port subtype as opposed to `Input` which identifies a block subtype. In contrast, in Massif the `InPort` and `InPortBlock` classes are used to refer to the port and block elements, respectively. Finally, MATLAB also provides data types such as Cell Arrays[2] and Structure Arrays[3] which Massif stores as plain Strings.

**From Simulink to EMF and vice-versa.** Massif provides four different ways to transform Simulink models into an EMF-compatible representation. This process is known as the *import* process. The import modes can affect performance of the process as they differ in the way the MATLAB/Simulink `ModelReference` blocks[4] are resolved: The *shallow* mode does not process the referenced model; the *deep* mode creates new `SimulinkModel` elements for each `ModelReference` block; the *flattening* model processes these blocks as `SubSystem` blocks; and the *referencing* mode processes `ModelReference` blocks as new EMF resources (once) and references them in the model.

The Massif *export* process consist on transforming the Simulink EMF-compatible representation into a MATLAB/Simulink file. This process can produce files with either `.slx` or `.mdl` extension.

## III. Live MATLAB/Simulink bridge

In this section we introduce the architecture and implementation of an approach that directly bridges MATLAB/Simulink models, including their MATLAB/Stateflow elements, through the on-the-fly translation of model management constructs into MATLAB commands. We choose the Epsilon model management framework to implement and evaluate our approach based on the connectivity facilities that it offers, which abstract-away the run-time model management constructs from the concrete modelling technology, and for the variety of model management languages in which the implementation becomes available. Other model management frameworks with similar connectivity facilities, such as ATL [22], could have been used instead. We refer to our implementation as the Simulink EMC driver, which is available under the Epsilon project[5]. [12] and its architecture is illustrated at the bottom compartment of Figure 3.

As discussed in subsection II-B, the Epsilon Model Connectivity (EMC) layer enables the uniform navigation and manipulation of models in any Epsilon model management language regardless of the model's underlying technology. Our implementation of the Simulink EMC driver is able to manage "live" MATLAB/Simulink models because it generates MATLAB commands executed on the model on-demand. To achieve this, the Simulink EMC driver connects to MATLAB's engine via the MATLAB Java API. To illustrate the on-the-fly translation approach, consider the EOL program below to be injected a model managed by the Simulink EMC driver at runtime.

```
Block.all.select(b|b.Name == 'MyBlock');
```

The `Block` type and its `Name` property would become available to the script through the use of on-the-fly translation of type and property getters into appropriate MATLAB commands. In other words, to retrieve all the `Block` model elements (i.e. `Block.all`), the following MATLAB command is submitted to the MATLAB engine for evaluation after the `?` placeholders are replaced by the name of the model and the `Block` keyword, in that order.

```
find_system(?, 'type', '?',...)
```

Then, the returned collection of block identifiers is internally managed by the Simulink EMC driver as a collection of `SimulinkBlock` instances, described in subsection III-A. By default the EOL `select` operator that follows iterates over the elements of the collection and filters by a condition but the Simulink EMC driver is required in the condition iterator to retrieve the `Name` property of a block. For that, the MATLAB statement below is submitted to the MATLAB engine after replacing the placeholder with block's identifier.

```
get_param(?, 'Name')
```

[2]Indexed data containers that can store any type of data.
[3]Groups of data in containers that store any type of data
[4]Blocks that represent a reference to another model

[5]https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/plugins/org.eclipse.epsilon.emc.simulink

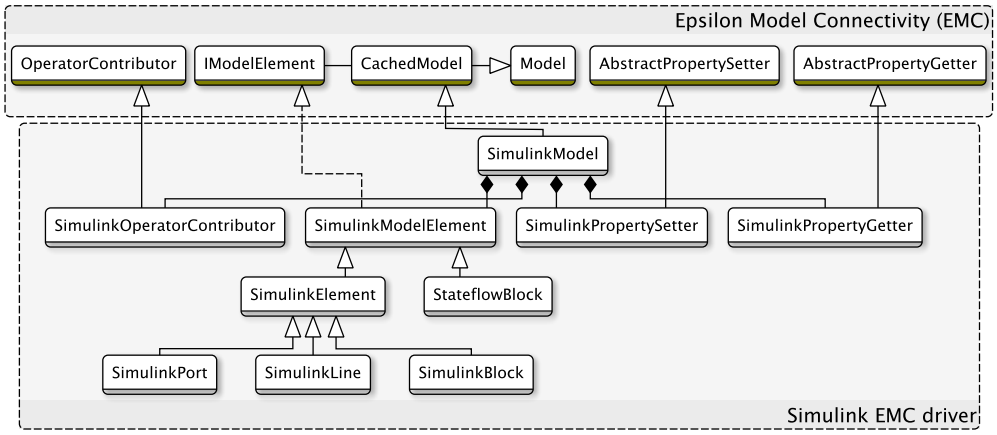


Fig. 3. Epsilon Simulink EMC driver architecture

### A. Model

The Simulink EMC driver considers a Simulink file as a model. A model is managed as an instance of the `SimulinkModel` class (see Figure 3). An instance of `SimulinkModel` defines the behaviour of inherited methods from the `CachedModel` and `Model` classes of the EMC layer, which describe how the model will perform CRUD operations on its *owned* model elements and the model itself.

The `SimulinkModel` class determines how to *load* and *dispose* the Simulink model, before and after the execution of a model management program e.g. validation, navigation. When the model is loaded, the Simulink EMC driver establishes a communication with the MATLAB engine.

### B. Model Elements

The `SimulinkModel` manages elements that inherit from the `SimulinkModelElement` class which can be either `SimulinkElement` or `StateflowBlock`.

Instances of `SimulinkElement` can be further decomposed into `SimulinkBlock`, `SimulinkPort` and `SimulinkLine` and handle MATLAB/Simulink elements of type `Block`, `Port` and `Line`, respectively. In Epsilon, the union of an element's super types and of its concrete type is referred to as the element's *kinds*. The Simulink EMC driver considers the MATLAB/Simulink subtype as the model element concrete type and considers both the MATLAB's *subtype* and *type* as the element's kinds. For example, a MATLAB/Simulink element of type `Block` and subtype `Gain` would be considered by the Simulink EMC driver as a model element of type `Gain` and of `Gain` and `Block` kinds.

MATLAB Simulink model elements provide different ways to be identified (e.g. *path*, *id*, *handle*). The Simulink EMC driver uses as identifier their *handle* property which is a non-persistent session-based immutable identifier of type Double. MATLAB queries return *handles* or *paths*, we chose *handles* as the *paths* are sensitive to the containment location of a model element and *ids* are only available to the latests MATLAB versions.

**Create.** The `SimulinkModel` instance manages the creation of block model elements. When the `new` reserved word is called in an EOL script the method `createInstance(type:String)` of the `SimulinkModel` (inherited from the `Model` class in the EMC layer) is invoked. To create blocks, this method requests the execution of the `add_block` MATLAB function, which requires the path of the library block to use for instantiation. Listing 4 shows the creation of `Sum` and `SubSystem` blocks in EOL using their library block path. The use of the back-tick (`) is required when a type identifier contains spaces. These blocks are created at the top level of the Simulink Model but can later be placed elsewhere by changing their parent.

```
1  var sum = new `simulink/Math Operations/Sum`;
2  var subsystem = new `simulink/Ports & Subsystems/
       Subsystem`;
```

Listing 4. Model element creation

There is no equivalent `add_port` function in MATLAB to create port model elements. In contrast, the `add_line` function which creates lines, requires the source and target ports to be connected. The Simulink EMC driver does not allow the direct creation of lines through a statement such as `new Line();` or `new signal();`. Instead, it creates them through the use of "linking" methods that may specify the source and/or target ports to be connected. For example, provided a model with the blocks in Figure 1 but no lines, these can be created with the following EOL program:

```
pulse.link(gain);
gain.linkTo(integrator, 1);
integrator.linkFrom(outport1, 1);
integrator.linkFrom(outport2, 2);
```

Listing 5. Linking methods for block elements

**Delete.** When an EOL statement uses the `delete` reserved word, as in Listing 6, the `SimulinkModel` instance calls the method `deleteElementInModel(element: SimulinkModelElement)` inherited from the `Model` class. This method retrieves the model element identifier (i.e. its handle) to request the evaluation of the `delete_block` or `delete_line` MATLAB functions[6].

```
1  delete sum;
2  delete subsystem;
```

Listing 6. Model element deletion

**Read.** Listing 7 illustrates different model element collection mechanisms in EOL, given a model `M`. The `allContents`

[6]There is no equivalent `delete_port` function

and `all` keywords invoke `SimulinkModel` methods that retrieve model elements. These methods, inherited from the Model class, request to the MATLAB API the evaluation of variations of the `find_system` MATLAB function and their results are mapped to lazy collections of `SimulinkModelElement` objects.

The `all` keyword (lines 1-3) triggers the execution of the `getAllOfKindFromModel(kind:String)` method which assumes the `kind` argument is either `Block`, `Line` or `Port`. The submitted MATLAB command looks for elements of a *type* e.g. `find_system(model,'type','Port')`. If the `kind` argument is different to `Block`, `Line` or `Port` (lines 4-5), then the `SimulinkModel` will look by MATLAB subtype. The `allContents()` method in EOL (line 6) invokes the `allContentsFromModel()` method which simply aggregates results of collections by supertype (i.e. `Block`, `Port`, `Line`), including Stateflow blocks.

```
1  var blocks = M!Block.all();
2  var lines = M!Line.all();
3  var ports = M!Port.all();
4  var sums = M!Sum.all();
5  var subsystems = M!SubSystem.all();
6  M.allContents();
```

Listing 7. Retrieval of model elements

**Update.** The `SimulinkModel` delegates to instances of `SimulinkPropertyGetter` and `SimulinkPropertySetter` the retrival and modification of model element properties. In turn, this classes request the evaluation of the `get_param` or `set_param` MATLAB functions when EOL getters and setters are invoked. Lines 1 and 3 in Listing 8 are examples of EOL property setters while lines 2, 4 and 5 are examples of EOL property getters.

```
1  subsystem.name = "Controller";
2  var subsystemName = subsystem.name;
3  sum.description = "Sum block";
4  var sumDescription = sum.description;
5  var inportHandles = subsystem.LineHandles.Inport;
```

Listing 8. Get and set model element properties

In the particular case of line 5, the property `LineHandles` returns a Structured Array, which is a MATLAB-specific type that represents an array of key-value pairs. In MATLAB, their values are retrieved using the `getfield(element,property)` function. The Simulink EMC driver can identify these types and navigates them as any other property. In the example, the value of the `Inport` key is retrieved.

**Methods.** Our Simulink EMC driver provides helper methods, such as the linking mechanisms in Listing 5, to facilitate common model and model element operations. Other methods such as `getType()`, `getParent()` and `getChildren()` are also available. Nevertheless, MATLAB provides many more functions for its Simulink and Stateflow model elements that would be challenging to individually include in the EMC driver. To deal with this, when an unknown method in EOL is called on the model or its elements the following strategy is applied.

Many MATLAB/Simulink API functions at model and model element level have a common syntax which takes the model element as first argument:

```
method_name(element, arg0, ..., argN)
```

while model element operations in EOL are executed as instance methods using the following syntax:

```
element.methodName(arg0, ..., argN);
```

To enable the execution of these unknown MATLAB functions, the EMC driver dynamically translates the method as a MATLAB command and submits it to the MATLAB engine for evaluation. For example, the EOL statements below

```
subsystem.find_mdlrefs();
subsystem.find_mdlrefs("AllLevels",true);
```

become the following MATLAB commands, where `subsystem` represents the MATLAB identifier of the block:

```
find_mdlrefs(subsystem)
find_mdlrefs(subsystem,'AllLevels',true)
```

### C. Stateflow

Our Simulink EMC driver can also manage Stateflow model elements. MATLAB handles these blocks different from Simulink elements. Figure 4 shows some Stateflow model elements contained under a Simulink `Chart` block. The figure contains two states named ON and OFF that represent operating modes of a system, and one transition, named E1, that goes from one state to the other. The arrow on the left is not a transition.

In MATLAB, all Stateflow types are preceded by the `Stateflow` keyword and a period. Our driver uses the same convention to differentiate them from `SimulinkElements`.

MATLAB/Stateflow model elements need a parent to be instantiated. For example, an element of type `Stateflow.State` in MATLAB is created using the `Stateflow.State(chart)` MATLAB statement, where `chart` is a reference to a `Stateflow.Chart` element used as parent. In EOL this state can be created with the following statement `new `Stateflow.State`(chart)`. In addition, the Simulink EMC driver can delay the instantiation of Stateflow elements until the parent is resolved. In other words, a state placeholder is created when using the `new `Stateflow.State`` statement —with no parent, and have its properties updated but only have the state instantiatiated and updated in MATLAB when its `parent` property is assigned.

In the Simulink EMC driver, Stateflow elements are managed by the `StateflowBlock` class. Stateflow model elements in MATLAB use a syntax closer to EOL to get and set their properties. For example, the name of a `Stateflow.State` element can be retrieved with the statement `element.Name` and have its value set with `element.Name = 'NewName'`. Assuming there is a `chart` variable of type `Stateflow.Chart`, the elements in Figure 4 can be created with the following EOL program:

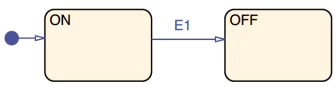


Fig. 4. Example of MATLAB/Stateflow model elements

```
var on = new `Stateflow.State`;
on.Name = "ON";
on.parent = chart;
var off = new `Stateflow.State`(chart);
off.Name = "OFF";
var tOnOff = new `Stateflow.Transition`(chart);
tOnOff.Source = on;
tOnOff.Destination = off;
tOnOff.LabelString = "E1";
```

## IV. Evaluation

This section evaluates the execution-time performance of two approaches to bridge MATLAB/Simulink models in a model management framework. The first approach consist in the use of the Simulink EMC driver to manage models in the Epsilon model management framework. The second approach consist on the use of Massif facilities to transform Simulink models into an EMF-compatible representation. Epsilon provides an EMF EMC driver able to read and write arbitrary EMF-based models which we use to manage the those produced by Massif. In the following, we refer to the first approach as *live* —since it manipulates the actual Simulink model, and to the second one as *Massif/EMF* —as it uses the Massif import facilities to produce their EMF-compatible representation.

Epsilon supports model element caching through a `CachedModel` abstraction that both the Simulink EMC driver and the EMF EMC driver reuse. We evaluate both approaches with these facilities enabled and disabled.

### A. Experiment setup

In order to evaluate the model management of Simulink models with Massif or the Simulink EMC driver, we compare the execution-time performance of the validation process of large Simulink models for each technology, which consists on the execution of EVL invariants that validate the structural properties of the model.

**Validation process.** EVL has a dedicated engine that consumes an EVL validation script and any number of models provided by EMC drivers of arbitrary modelling technology at runtime. An example of an EVL script is shown in Listing 9. This script contains an invariant (line 2) of type `critique` and name `BlockNameIsLowerCase` that is validated against all model elements of kind `Block` as specified by the `context` reserved word in line 1. Invariants may be of type `constraint` or `critique` depending on the severity level of their failure i.e. a `constraint` produces errors while a `critique` produces warnings. Line 3 shows the EOL statement that is executed on each of the block model elements which verifies that the name of the element is lowercase. The `self` reserved word is a reference to the current model element the invariant is acting on. If a given block fails the `check` statement, then `fix` elements become available if present in the invariant declaration. In the example, the `fix` in line 4 updates the element name to lowercase as indicated in line 7 by the `do` environment. The `title` of the fix (line 5) is just informative.

```
1  context Block {
2    critique BlockNameIsLowerCase {
3      check:  self.Name == self.Name.toLowerCase()
4      fix {
5        title: "Name to lower case"
6        do {
7          self.Name = self.Name.toLowerCase();
8        }
9      }
10   }
11 }
```

Listing 9. Sample EVL script with invariant 9 from Table II

Before the EVL engine can execute the validation script, the models involved must be loaded. When the EMF EMC driver is used to process an EMF model, the loading stage consist on the registration and resolution of the model and meta-model resources and packages. When the Simulink EMC driver is used to process a MATLAB/Simulink model file, the driver establishes the connection with the MATLAB engine and requests the model to be loaded in the engine. Once a model is loaded by the corresponding EMC driver, then the EVL engine can execute validations parsed from an EVL script against the model. In the following we consider the model loading and validation execution as two different stages of the validation process. These are depicted in Figure 5 as stages 1 and 2. In addition, for the Massif/EMF approach we consider the transformation of the model —from Simulink to EMF, as an additional stage of the validation process which we call the *import* stage (Stage 0 in Figure 5) after the Massif facilities that enable this transformation.

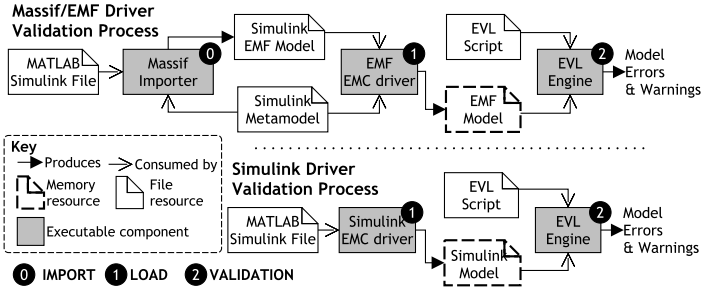


Fig. 5. Validation setup with no upfront transformation.

The implementation of the EMC drivers and the structure of the meta-model used in the EMF driver affect the way the model is navigated in EOL-based programs. Consequently, the EVL validation script cannot be reused *as-is* across approaches. To illustrate this, consider an EOL program that retrieves the `PortDimension` property of a `block` model element. Using the Simulink EMC driver to inject the model at runtime, the EOL statement below is able to retrieve this property.

```
block.PortDimension;
```

In contrast, when using the EMF driver with the Massif meta-model, the statement needs to be adapted since the `Block` class in the meta-model does not have a `PortDimension` attribute but instead has a `parameters` attribute containing a set of `Property` elements, one of them with the `PortDimension` identifier. In this case, the EOL statement becomes:

```
block.parameters.selectOne(p | p.name == "
    PortDimension").value;
```

We measure the execution-time performance of the different stages of the validation process i.e. (0) Simulink-to-EMF transformation, (1) model loading, and (2) model validation. Notice that: Stage 0 is only applicable to the *Massif/EMF* approach; Stage 1 is applicable to both approaches; and Stage 2 is applicable to both approaches with the caching facilities of the EMC drivers enabled or disabled.

Each stage of the validation process was executed 20 times with 5 warm-up iterations for each model. We used the Java Microbenchmark Harness (JMH) [23] tool to run these experiments on a quad core Intel Core i5-7200U CPU @ 2.5 GHz with 16GB of RAM. The Java Virtual Machine (64-Bit) was provided with up to 10GB of memory and ran Java 8 on JDK 1.8.0_152. All EMF-compatible models were generated using the *shallow* mode of the Massif import facilities which does not process external model references. The validation scripts and the Simulink models that were used in our experiments can be found in the examples of the Epsilon project[7].

**Validation scripts.** Equivalent EVL scripts are used to evaluate each approach. They are equivalent to the best of our knowledge as they are using (a) equivalent EVL *contexts* which may vary in naming across approaches (e.g. `Inport` vs. `InPortBlock`), (b) equivalent model element navigations (e.g. `self.parameters.selectOne(p|p.name == 'PortDimensions').value`) for the EMF EMC driver and `self.PortDimensions` for the Simulink EMC driver), and (c) equivalent way in which the constraint checks and guards are prescribed. Each script consists of 9 invariants (see Table II) inspired in model checks used by industrial partners but also intended to exercise the model through typical query language features [24] performed on signature model element types [1]. In Table II the *Kind* column refers to query checks inspired on well-formedness constraint categories used by the Train Benchmark [24], and the *Context* column refers to the EVL context, that is, the model element types on which the invariant is executed. Stateflow blocks were not included in the validation scripts as Massif does not support them.

The validation scripts for the *live* approaches used 96 lines of code (LOC) and that for the *Massif/EMF* approach used 110 LOC. The body of the invariants was written in the same amount of lines for both approaches (89 LOC) and the extra lines were related to helper operations.

**Model selection.** We used BigQuery [25] to find a list of Simulink files (*.slx) publicly available in GitHub that were larger than 1MB. Out of the 70 models found, we selected the first 7 models that could be translated into EMF in under 2 hours using Massif's import facilities. Table I shows the number of model elements of each type used in the validation. The number of `block` elements on the models ranged from 8628 to 9536.

The selected models had dependencies to proprietary Simulink libraries. For simplicity, we did not process any libraries in any approach.

[7]https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/examples/org.eclipse.epsilon.examples.emc.simulink.emf

| Size | Block | Inport | Outport | Goto | From | SubSystem |
|---|---|---|---|---|---|---|
| **1.112** | 8785 | 1373 | 1177 | 69 | 103 | 717 |
| **1.131** | 8628 | 1372 | 1167 | 62 | 93 | 740 |
| **1.133** | 8645 | 1372 | 1167 | 62 | 93 | 740 |
| **1.134** | 9536 | 1489 | 1269 | 38 | 57 | 861 |
| **1.135** | 8645 | 1372 | 1167 | 62 | 93 | 740 |
| **1.138** | 8651 | 1376 | 1177 | 62 | 93 | 745 |
| **1.141** | 8634 | 1374 | 1156 | 67 | 99 | 714 |

TABLE I
NUMBER OF ELEMENTS PER TYPE BY MODEL SIZE.

### B. Results

All validation invariants were executed in the same number of model elements for all approaches. Similarly, the results of the validation reported the same number of failed constraints on all approaches. The file size of the EMF models produced by the import stage are displayed in Figure 6.

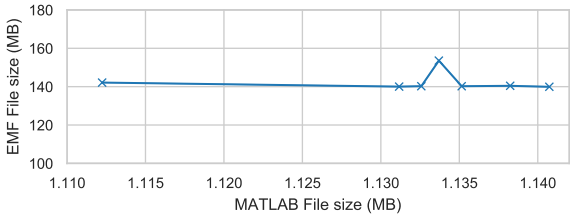


Fig. 6. Size of the imported EMF models against the original MATLAB files.

Figure 7 shows the whole validation process execution-time (in minutes) calculated using the average sum of each stage for each approach with and without caching. Figure 8 shows the execution time of each stage of the model validation process (in seconds and logarithmic scale) against the size of the MATLAB model files (in MB): Sub-figure (a) displays the duration distribution of Massif's *import* task (Stage 0) which transforms Simulink models into an EMF-compatible model. Similarly, Sub-figure (b) displays the duration distribution of the model loading task (Stage 1), required by both the EMF and Simulink EMC drivers. Sub-figure (c) displays the duration distribution of the model validation task (Stage 2) for bo

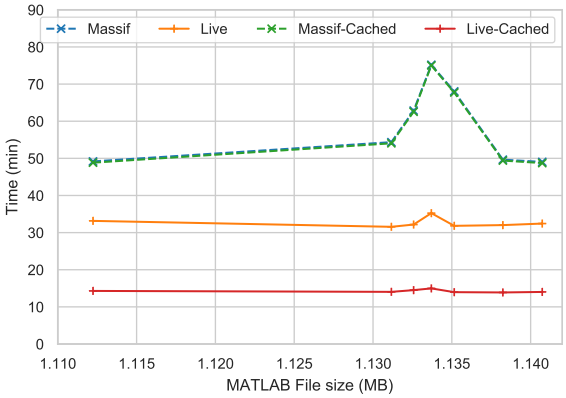


Fig. 7. Total execution-time duration (log-scale) against MATLAB file size. *Note that Massif and Massif-Cached overlap.*

Figure 8 shows that most of the performance overhead of the Massif/EMF approach happens at the import stage while most of the Simulink EMC driver overhead happens at the validation stage. The import stage of the Massif/EMF approach took

| # | Kind | Context | Description |
|---|---|---|---|
| 1 | PropertyCheck | Goto | TagVisibility property is local |
| 2 | NavigationAndFilter | From | There is a Goto block in scope with the name of the GotoTag property |
| 3 | PropertyCheck | Inport/InPortBlock | PortDimensions property should not be inherited (-1) |
| 4 | PropertyCheck | Outport/OutPortBlock | Description property is not null or empty |
| 5 | NavigationAndFilter | SubSystem | ForegroundColor property is green for all connected Inport blocks |
| 6 | TransitiveClosure | SubSystem | Subsystem is no more than three levels deep |
| 7 | VertexConnectivity | SubSystem | All outports are connected |
| 8 | LoopAbsence | SubSystem | No feedback. Outports do not connect to the same subsystem |
| 9 | PropertyCheck | Block | Block's name is in lower case |

TABLE II
EVALUATED INVARIANTS

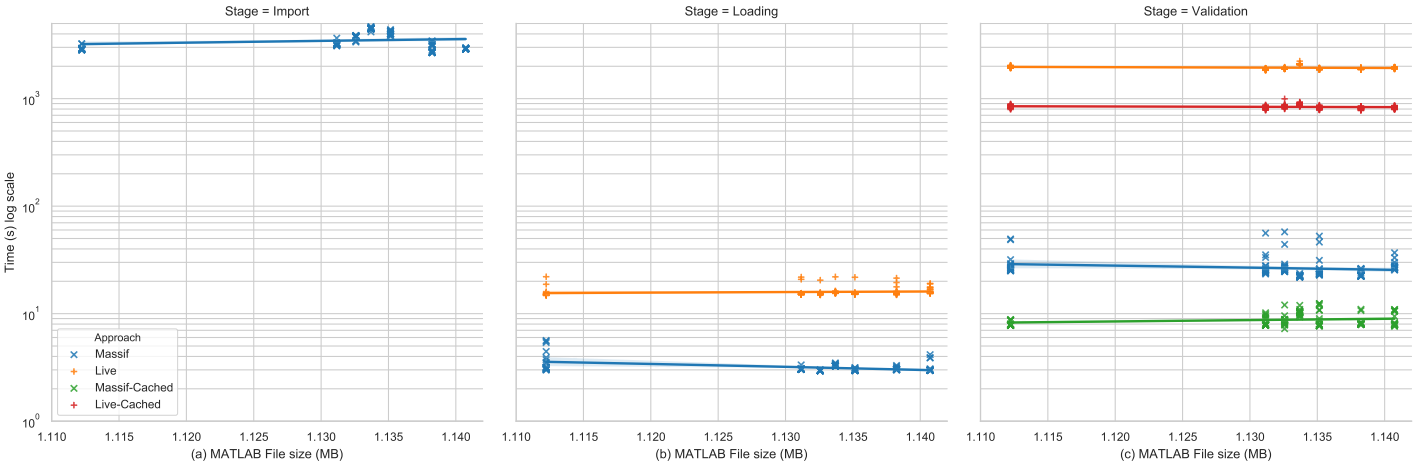


Fig. 8. Execution-time duration in log-scale against MATLAB/Simulink model file size per stage of the validation process.

between 4,486 and 2,911s to finish. The Massif/EMF approach achieved the loading stage in 2.95-3.63s while the Simulink EMC driver achieved it in 15.5-16.5s. The live approach was approximately 1 order of magnitude slower at the loading stage. In the validation stage, the Massif/EMF approach took between 22.4-28.9s while the Simulink EMC driver took 1,877-2,098s to complete. With caching facilities enabled in both drivers, the Massif/EMF approach took 8.10-10.2s while the Simulink EMC driver took 816-882s to finish. With and without caching, the live approach was approximately 2 orders of magnitude slower at the validation stage. The caching facilities improved the performance in the validation stage by 54.4-72.0% in the Massif/EMF approach and 55.3-58.0% in the live approach.

When we compare the overall performance, that is, the sum of the average execution per stage of the different models, we observe that the live approach improves the performance of the Massif/EMF approach by taking 70.7-80.0% less time when caching is enabled and by 32.6-53.2% with no caching.

### C. Threats to Validity

Our evaluation only tested the performance of one model management language (EVL). Moreover, the validation script were limited to read-only operations.

The sample of models may not be significant but was limited by the 2-hour cap imposed to the import stage. Our experiments would benefit from more diverse models with a broader range of sizes and more varied constraints.

## V. OBSERVATIONS AND LESSONS LEARNED

This section summarises our observations and lessons learned in the implementation of the Simulink EMC driver and our experiments.

**Performance.** Model validation processes generally involve several iterations of checking constraints and fixing errors, unless the model is correct to start with. For this reason, we consider the *live* approach more suitable for large models in development as it improved the overall performance by 80% in our experiments. In contrast, for large signed-off models that need to be extensively queried, the Massif/EMF approach is much suitable as the cost of the transformation to EMF is paid once and the validations are faster and the Simulink EMC driver differed by 2 orders of magnitude. The performance of the Simulink EMC driver is likely influenced by the overhead of calls to the MJ-API. To improve performance, operations on collections of model elements could be optimised to submit MATLAB/Simulink commands that execute bulk operations.

**Meta-model fidelity.** One of the findings of this work is that the MATLAB's API provides sufficiently fine-grained facilities to support on-the-fly translation and execution of OCL-like queries on Simulink models and their Stateflow components and even provide support for MATLAB-specific data types. We have discussed in subsection II-C how the Simulink Ecore meta-model provided by Massif uses different names to refer to MATLAB/Simulink model elements. In contrast, model element types used in the Simulink EMC driver are closer to those managed by the MATLAB command line interface. In addition, our Simulink EMC driver provides support for Stateflow elements. Moreover, the MATLAB specific data

types can be manipulated as such in the Simulink EMC driver while in their EMF-counterpart they are managed as strings.

**Model file size and model elements.** In Figure 6 we observe that the size of the EMF model produced by Massif is much larger than the original MATLAB/Simulink (.slx) files. This is partly due to .slx being a compressed file format. As Table I shows, the size of the MATLAB/Simulink file is not directly proportional to the number of `Block`[8] elements in the model. In contrast, the size of the EMF model file seems to be related to the number of block elements, which would explain the peak on the EMF file size with the MATLAB/Simulink model with the largest number of block elements.

## VI. Related Work

It is often desirable to have a common framework to manage models from heterogeneous modelling technologies. Examples of those frameworks are traceability tools such as Capra [26] and Yakindu [27], which need to be able to read models used at different stages of de the development process in order to create and manage traces among their model elements. Other examples include model management frameworks such as Epsilon [12] and ATL [22], which offer a subset of task-specific languages for model navigation, validation, model-to-model or model-to-text transformations, etc. and which are able to interact with a number of models of arbitrary underlying technologies. Similarly, Software-as-a-Service (SaaS) modelling platforms are used to foster heterogeneous models and execute model management scripts as a service. MDEForge [28] is an example of a SaaS platform though it is currently limited to ATL model-to-model transformations and EMF models [29].

When model management frameworks do not offer support for a specific modelling technology such as Simulink, import and export facilities can be used to translate the models into a supported format. Possibly for protective reasons, proprietary modelling tools don't always offer exporting facilities into open modelling formats such as XMI. MATLAB, in particular, does not offer any export or import facilities for Simulink Models with other open-source modelling formats. To solve this feature gap, the open-source Massif project led the development of import and export facilities between EMF and Simulink models. Massif internally uses MATLAB's command line interface to parse the Simulink models and populate their EMF representation and vice-versa.

The Open Services for Lifecycle Collaboration (OSLC) [30] is an initiative that aims at simplifying the software tool integration problem among proprietary tools. Built atop the W3C Resource Description Framework (RDF), Linked Data, and the REST architecture, OSLC provides set of specifications targeted at different aspects of application and product life cycle management. Nevertheless, the comprehensiveness of the information exposed by these services is at the discretion of service provider. MATLAB does not officially provide an OSLC interface for its Simulink models, although the Eclipse Lyo [31] project provides a Simulink OSLC adaptor [32] for MATLAB version R2013b and also Massif [11] provides an OSLC adaptor for their EMF-compatible representations [33].

Transformations from SysML to Simulink models (and vice-versa) have motivated several research works such as [8, 9, 10]. [8], [9] and [10] wrote model-to-text transformations with Acceleo [34] to produce MATLAB scripts that on execution created the Simulink model. More specifically, [9] generated several MATLAB scripts to populate different parts of the Simulink model, [10] proposed the use of a UML profile to annotate the SysML models before the MATLAB code generation, and [8] suggested that to go back from Simulink to SysML the creation of a MATLAB script to parse Simulink models and produce an XML-based SysML model description file. As the previous works are dealing with a specific transformation (SysML to Simulink scripts) they are not very reusable nor allow the actual management of Simulink models. In this regard, the Massif project facilities brought the possibility of managing an actual EMF-compatible Simulink model representation in a broader range of model management scenarios at the expense of having to co-manage both artefacts. In contrast, our approach uses the modelling technology API to translate high-level CRUD operations at *model type* level on-demand in a similar fashion to other Epsilon bridges such as [20, 35].

## VII. Conclusions and Future Work

We have proposed an approach to bridge MATLAB/Simulink models with the Epsilon model management framework that uses on-the-fly translation of on-demand model management constructs into MATLAB commands. Our approach eliminates the need for a transformation into an EMF-compatible representation and for the co-evolution of this EMF-counterpart that current solutions require, and enables complete Simulink model management including Stateflow components. We have evaluated our bridge against an approach that requires the Simulink model upfront transformation into EMF using Massif facilities. Our experiments measured the execution time performance of a model validation process performed on a sample of the largest publicly available Simulink models in GitHub (up to 1.141 MB and 9536 blocks) using both approaches. Our evaluation results support the claim that the transformation of large Simulink models into an EMF-compatible representation can be very expensive and shows that our bridge can reduce the effect of the transformation in the time required for the validation process by up to 80%.

**Future Work.** We plan to extend the Simulink EMC driver to support MATLAB toolboxes such as the Simulink Requirements. In addition, the model element collection and selection operations could be optimised through parallelisation, pagination and more efficient MATLAB queries. The evaluation can be extended to include models from a broader size range, the application of validation fixes, the inclusion of Stateflow elements and the performance of the EMF-to-Simulink transformation when EMF models constantly change.

[8] `Inport`, `Outport`, `Goto`, `From` and `SubSystem` are all subtypes of `Block`

## ACKNOWLEDGMENTS

The work in this paper was partially supported by Innovate UK and the UK aerospace industry through the SECT-AIR project; by the Engineering and Physical Sciences Research Council (EPSRC) through the National Productivity Investment Fund (NPIF) in partnership with Rolls-Royce under Grant No.: EP/R512230/1; and by the Mexican National Council for Science and Technology (CONACyT) under Grant No.: 602430/472773.

## REFERENCES

[1] M. Bender, K. Laurin, M. Lawford, V. Pantelic, A. Korobkine, J. Ong, B. Mackenzie, M. Bialy, and S. Postma, "Signature required: Making Simulink data flow and interfaces explicit," *Science of Computer Programming*, vol. 113, pp. 29–50, 2015.

[2] V. Pantelic, S. Postma, M. Lawford, A. Korobkine, B. Mackenzie, J. Ong, and M. Bender, "A Toolset for Simulink - Improving Software Engineering Practices in Development with Simulink," *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pp. 50–61, 2015.

[3] V. Pantelic, S. Postma, M. Lawford, M. Jaskolka, B. Mackenzie, A. Korobkine, M. Bender, J. Ong, G. Marks, and A. Wassyng, "Software engineering practices and Simulink: bridging the gap," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 1, pp. 95–117, 2018.

[4] The Eclipse Foundation, "Eclipse Modeling Framework." [Online]. Available: http://www.eclipse.org/emf

[5] D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige, "Eugenia: towards disciplined and automated development of GMF-based graphical model editors," *Software and Systems Modeling*, vol. 16, no. 1, pp. 229–255, 2017.

[6] The Eclipse Foundation, "Papyrus." [Online]. Available: https://www.eclipse.org/papyrus/

[7] PolarSys, "Capella." [Online]. Available: http://www.polarsys.org/capella/

[8] A. Sindico, M. Di Natale, and G. Panci, "Integrating SysML with Simulink using open-source model transformations," *SIMULTECH 2011 - Proceedings of 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pp. 45–56, 2011.

[9] M. D. Natale and F. Chirico, "An MDA Approach for the Generation of Communication Adapters Integrating SW and FW Components from Simulink," *Model-Driven Engineering Languages and Systems*, vol. 8767, pp. 353–369, 2014.

[10] B. Chabibi, A. Douche, A. Anwar, and M. Nassar, "Integrating SysML with simulation environments (Simulink) by model transformation approach," *Proceedings - 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2016*, pp. 148–150, 2016.

[11] Viatra, "Massif: MATLAB Simulink Integration Framework for Eclipse." [Online]. Available: https://github.com/viatra/massif

[12] The Eclipse Foundation, "The Epsilon Project." [Online]. Available: https://www.eclipse.org/epsilon/

[13] MathWorks, "MATLAB Simulink." [Online]. Available: https://www.mathworks.com/products/simulink.html

[14] Mathworks, "MATLAB Stateflow." [Online]. Available: https://uk.mathworks.com/products/stateflow.html

[15] MATLAB & Simulink, "Create Simple Model." [Online]. Available: https://uk.mathworks.com/help/simulink/gs/create-a-simple-model.html

[16] Mathworks, "MATLAB Java Engine API Summary." [Online]. Available: https://uk.mathworks.com/help/matlab/matlab_external/java-api-summary.html

[17] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon Object Language (EOL)," *Lecture Notes in Computer Science*, vol. 4066 LNCS, pp. 128–142, 2006.

[18] D. S. Kolovos, R. F. Paige, and F. A. Polack, "On the evolution of OCL for capturing structural constraints in modelling languages," *Lecture Notes in Computer Science*, vol. 5115 LNCS, pp. 204–218, 2009.

[19] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," *Lecture Notes in Computer Science*, vol. 5063 LNCS, pp. 46–60, 2008.

[20] M. Francis, D. S. Kolovos, N. Matragkas, and R. F. Paige, "Adding spreadsheets to the MDE toolkit," in *Lecture Notes in Computer Science*, vol. 8107 LNCS, 2013, pp. 35–51.

[21] Massif, "Massif Simulink Ecore Documentation," Tech. Rep., 2015. [Online]. Available: https://github.com/viatra/massif/wiki/pdf/massif-simulink-ecore-doc.pdf

[22] The Eclipse Foundation, "The ATLAS Transformation Language Project." [Online]. Available: https://www.eclipse.org/atl/

[23] OpenJDK, "Java Microbenchmark Harness." [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/

[24] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró, "The Train Benchmark: cross-technology performance evaluation of continuous model queries," *Software & Systems Modeling*, pp. 1–29, jan 2017.

[25] Google, "BigQuery." [Online]. Available: https://bigquery.cloud.google.com/

[26] S. Maro and J.-P. Steghofer, "Capra: A Configurable and Extendable Traceability Management Tool," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*. IEEE, sep 2016, pp. 407–408.

[27] itemis AG, "Yakindu Traceability." [Online]. Available: https://www.itemis.com/en/yakindu/traceability/

[28] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, "MDEForge: An extensible Web-based modeling platform," *CEUR Workshop Proceedings*, vol. 1242, no. 619583, pp. 66–75, 2014.

[29] J. Di Rocco, D. Di Ruscio, A. Pierantonio, J. S. Cuadrado, J. De Lara, and E. Guerra, "Using ATL transformation services in the MDEForge collaborative modeling platform," *Lecture Notes in Computer Science*, vol. 9765, pp. 70–78, 2016.

[30] "Open Services for Lifecycle Collaboration (OSLC)." [Online]. Available: http://open-services.net/

[31] The Eclipse Foundation, "The Lyo Project." [Online]. Available: https://www.eclipse.org/lyo/

[32] The Eclipse Foundation, "Lyo Simulink Adapter." [Online]. Available: https://wiki.eclipse.org/Lyo/Simulink

[33] A. Horvath, I. Rath, and R. Rizzi Starr, "Massif - the love child of Matlab Simulink and Eclipse — EclipseCon NA 2015," 2015. [Online]. Available: http://www.eclipsecon.org/na2015/session/massif-love-child-matlab-simulink-and-eclipse.html

[34] The Eclipse Foundation, "The Acceleo Project." [Online]. Available: https://www.eclipse.org/acceleo/

[35] A. Zolotas, H. H. Rodriguez, D. S. Kolovos, R. F. Paige, and S. Hutchesson, "Bridging Proprietary Modelling and Open-Source Model Management Tools: The Case of PTC Integrity Modeller and Epsilon," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, sep 2017, pp. 237–247.